Danijar Hafner

构造你的TensorFlow模型

Apr 26, 2016

在TensorFlow中定义你的模型可以轻松地导致一个巨大的代码墙。如何以可读和可重复使用的方式构建代码？

定义计算图

从每个模型开始一个类是明智的。该类的接口是什么？通常，您的模型连接到一些输入数据和目标占位符，并提供训练，评估和推理的操作。

```
class Model:

    def __init__(self, data, target):
        data_size = int(data.get_shape()[1])
        target_size = int(target.get_shape()[1])
        weight = tf.Variable(tf.truncated_normal([data_size, targe
        bias = tf.Variable(tf.constant(0.1, shape=[target_size]))
        incoming = tf.matmul(data, weight) + bias
        self._prediction = tf.nn.softmax(incoming)
        cross_entropy = -tf.reduce_sum(target, tf.log(self._predic
        self._optimize = tf.train.RMSPropOptimizer(0.03).minimize(
        mistakes = tf.not_equal(
            tf.argmax(target, 1), tf.argmax(self._prediction, 1))
        self._error = tf.reduce_mean(tf.cast(mistakes, tf.float32)

    @property
    def prediction(self):
```

```
        return self._prediction

    @property
    def optimize(self):
        return self._optimize

    @property
    def error(self):
        return self._error
```

这是基本的，如何在TensorFlow代码库中定义模型。然而，它有一些问题。最值得注意的是，整个计算图是在单个函数中定义的，构造函数。这既不是特别可读也不可重用。

使用属性

将代码分解为函数就不起作用，因为每次调用函数时，计算图都将被新的代码扩展。因此，我们必须确保只有在函数被调用时才将操作第一次添加到计算图中。这基本上是延迟加载。

```
class Model:

    def __init__(self, data, target):
        self.data = data
        self.target = target
        self._prediction = None
        self._optimize = None
        self._error = None

    @property
    def prediction(self):
        if not self._prediction:
            data_size = int(self.data.get_shape()[1])
            target_size = int(self.target.get_shape()[1])
```

```
            weight = tf.Variable(tf.truncated_normal([data_size, t
            bias = tf.Variable(tf.constant(0.1, shape=[target_size
            incoming = tf.matmul(self.data, weight) + bias
            self._prediction = tf.nn.softmax(incoming)
        return self._prediction

    @property
    def optimize(self):
        if not self._optimize:
            cross_entropy = -tf.reduce_sum(self.target, tf.log(sel
            optimizer = tf.train.RMSPropOptimizer(0.03)
            self._optimize = optimizer.minimize(cross_entropy)
        return self._optimize

    @property
    def error(self):
        if not self._error:
            mistakes = tf.not_equal(
                tf.argmax(self.target, 1), tf.argmax(self.predicti
            self._error = tf.reduce_mean(tf.cast(mistakes, tf.floa
        return self._error
```

这比第一个例子好得多。你的代码现在被构造成可以单独关注的函数。但是，由于延迟加载逻辑，代码仍然有点臃肿。让我们看看如何改进这一点。

Lazy Property Decorator

Python是一种非常灵活的语言，可以自定义装饰器。那么让我来告诉你如何从上一个例子中删除冗余代码。我们将使用一个像@property这样的装饰器，但只能对该函数进行一次评估。它将结果存储在以装饰函数命名的成员中（前为前缀），并在任何后续调用中返回此值。

```
import functools

def lazy_property(function):
    attribute = '_cache_' + function.__name__

    @property
    @functools.wraps(function)
    def decorator(self):
        if not hasattr(self, attribute):
            setattr(self, attribute, function(self))
        return getattr(self, attribute)

    return decorator
```

Using this decorator, our example simplifies to the code below.

```
class Model:

    def __init__(self, data, target):
        self.data = data
        self.target = target
        self.prediction
        self.optimize
        self.error

    @lazy_property
    def prediction(self):
        data_size = int(self.data.get_shape()[1])
        target_size = int(self.target.get_shape()[1])
        weight = tf.Variable(tf.truncated_normal([data_size, targe
        bias = tf.Variable(tf.constant(0.1, shape=[target_size]))
        incoming = tf.matmul(self.data, weight) + bias
        return tf.nn.softmax(incoming)

    @lazy_property
    def optimize(self):
        cross_entropy = -tf.reduce_sum(self.target, tf.log(self.pr
        optimizer = tf.train.RMSPropOptimizer(0.03)
        return optimizer.minimize(cross_entropy)
```

```
    @lazy_property
    def error(self):
        mistakes = tf.not_equal(
            tf.argmax(self.target, 1), tf.argmax(self.prediction,
        return tf.reduce_mean(tf.cast(mistakes, tf.float32))
```

注意，我们在构造函数中提到了属性。这种方式可以保证完整的计算图由我们运行tf.initialize_variables（）时被定义

Organizing the Graph with Scopes

我们现在有一个清晰的方法来定义代码中的模型，但是生成的计算图仍然很拥挤。如果你将计算图可视化，它将包含很多相互关联的小节点。解决方案是用tf.name_scope（'name'）或tf.variable_scope（'name'）包装每个函数的内容。然后节点将在图中分组在一起。但是我们调整我们以前的装饰器自动做：

```
import functools

def define_scope(function):
    attribute = '_cache_' + function.__name__

    @property
    @functools.wraps(function)
    def decorator(self):
        if not hasattr(self, attribute):
            with tf.variable_scope(function.__name):
                setattr(self, attribute, function(self))
        return getattr(self, attribute)

    return decorator
```

我给装饰器一个新的名字，因为它具有特定于TensorFlow的功能，除了lazy caching。除此之外，该模型看起来与前一个模型相同。

我们可以更进一步地，使@define_scope装饰器将参数转发到tf.variable_scope()，例如定义作用域的默认初始化程序。如果你有兴趣，请查看下面我整理的全部例子。

我们现在可以以结构化和紧凑的方式定义模型，从而产生有组织的计算图。

TensorFlow Scope Decorator

blog_tensorflow_scope_decorator.py

```python
# Working example for my blog post at:
# https://danijar.github.io/structuring-your-tensorflow-models
import functools
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data


def doublewrap(function):
    """
    A decorator decorator, allowing to use the decorator to be used without
    parentheses if not arguments are provided. All arguments must be optional.
    """
    @functools.wraps(function)
    def decorator(*args, **kwargs):
        if len(args) == 1 and len(kwargs) == 0 and callable(args[0]):
            return function(args[0])
        else:
            return lambda wrapee: function(wrapee, *args, **kwargs)
    return decorator


@doublewrap
def define_scope(function, scope=None, *args, **kwargs):
    """
    A decorator for functions that define TensorFlow operations. The wrapped
    function will only be executed once. Subsequent calls to it will directly
    return the result so that operations are added to the graph only once.

    The operations added by the function live within a tf.variable_scope(). If
    this decorator is used with arguments, they will be forwarded to the
    variable scope. The scope name defaults to the name of the wrapped
    function.
    """
    attribute = '_cache_' + function.__name__
    name = scope or function.__name__
    @property
    @functools.wraps(function)
    def decorator(self):
        if not hasattr(self, attribute):
            with tf.variable_scope(name, *args, **kwargs):
                setattr(self, attribute, function(self))
        return getattr(self, attribute)
    return decorator


class Model:

    def __init__(self, image, label):
        self.image = image
        self.label = label
        self.prediction
        self.optimize
        self.error

    @define_scope(initializer=tf.contrib.slim.xavier_initializer())
    def prediction(self):
        x = self.image
        x = tf.contrib.slim.fully_connected(x, 200)
        x = tf.contrib.slim.fully_connected(x, 200)
```

```
        x = tf.contrib.slim.fully_connected(x, 10, tf.nn.softmax)
        return x

    @define_scope
    def optimize(self):
        logprob = tf.log(self.prediction + 1e-12)
        cross_entropy = -tf.reduce_sum(self.label * logprob)
        optimizer = tf.train.RMSPropOptimizer(0.03)
        return optimizer.minimize(cross_entropy)

    @define_scope
    def error(self):
        mistakes = tf.not_equal(
            tf.argmax(self.label, 1), tf.argmax(self.prediction, 1))
        return tf.reduce_mean(tf.cast(mistakes, tf.float32))


def main():
    mnist = input_data.read_data_sets('./mnist/', one_hot=True)
    image = tf.placeholder(tf.float32, [None, 784])
    label = tf.placeholder(tf.float32, [None, 10])
    model = Model(image, label)
    sess = tf.Session()
    sess.run(tf.initialize_all_variables())

    for _ in range(10):
      images, labels = mnist.test.images, mnist.test.labels
      error = sess.run(model.error, {image: images, label: labels})
      print('Test error {:6.2f}%'.format(100 * error))
      for _ in range(60):
        images, labels = mnist.train.next_batch(100)
        sess.run(model.optimize, {image: images, label: labels})


if __name__ == '__main__':
  main()
```